The Unison Manual

Roberto Castañeda Lozano

Contents

Introduction

You are reading the manual of Unison: a simple, flexible, and potentially optimal open-source tool that performs integrated register allocation and instruction scheduling using constraint programming.

Unison can be used as an alternative or as a complement to the algorithms applied by standard compilers such as GCC and LLVM. Unison is particularly easy to integrate with the latter as a driver is already available (see Chapter 3 for details).

This manual is divided into two main parts: Part I (Chapters 1 to 3) is devoted to the use of Unison, while Part II (Chapters 4 to 7) deals with its development and extension.

Chapter 1 discusses licensing aspects and provides contact information. Chapter 2 contains instructions to download, build, install, and test Unison. Chapter 3 describes how to use the LLVM driver.

Chapter 4 outlines the architecture of Unison. Chapter 5 describes the intermediate representation (*Unison IR*). Chapter 6 formulates the combinatorial model that lies at the core of the Unison approach. Chapter 7 provides information about how processors are described in Unison.

Appendix A provides references for further reading and other sources of documentation.

Part I

Using Unison

Chapter 1

License, Contact, and Acknowledgments

Unison is developed at the Swedish Institute of Computer Science in collaboration with KTH Royal Institute of Technology in Stockholm, Sweden.

Unison and the Unison Driver for LLVM are released under the BSD3 open-source license:

Unison includes code from the following projects:

- Graphalyze (in some graph algorithms of the `unison` package). The code is licensed under BSD2.

- JsonCpp (used by the presolver and solver to parse their input). The code is licensed under MIT.
- Erik Ekström's master's thesis (in parts of the presolver). The code is licensed under BSD3 but the copyright is held by Erik Ekström.
- Mikael Almgren's master's thesis (in parts of the presolver). The code is licensed under BSD3 but the copyright is held by Mikael Almgren.

The Unison Driver is built on top of the LLVM Compiler Infrastructure which is licensed under the University of Illinois/NCSA Open Source License, see the LLVM website for details.

Furthermore, Unison makes extensive use of other open-source components, including:

- Gecode
- Qt
- Graphviz
- Haskell Platform
- Various Haskell packages (see the `Build-Depends` field in this and this package descriptions)

For further license detail on these components, please check their websites.

Unison is designed, developed, and maintained by

- Roberto Castañeda Lozano (rcas@acm.org)
- Mats Carlsson (mats.carlsson@ri.se)
- Gabriel Hjort Blindell (ghb@kth.se)
- Christian Schulte (cschulte@kth.se)

Other people have also collaborated in the development of Unison:

- Özgür Akgün
- Mikael Almgren
- Noric Couderc
- Frej Drejhammar
- Erik Ekström
- Bevin Hansson
- Jan Tomljanović
- Kim-Anh Tran

Chapter 2

Getting Started

Unison has to be built from source as we do not yet provide precompiled packages. The tool is known to work on Linux and it might work on other platforms such as macOS and Windows as all software dependencies claim to be portable across these.

Building, installing, and testing Unison in Linux is relatively straightforward. Just take the following steps.

2.1 Downloading

Unison is hosted by GitHub. The easiest way to access its source code (including the history) is by running:

```
git clone https://github.com/unison-code/unison.git
```

2.2 Prerequisites

Unison has the following direct dependencies:

- Stack
- Qt (version 5.x)
- Graphviz library
- Gecode (version 6.0.0)

To get the first three dependencies in Debian-based distributions, just run:

```
apt-get install haskell-stack qtbase5-dev libgraphviz-dev
```

Upgrade Slack after installing it:

```
stack upgrade
```

The source of Gecode can be fetched with:

```
wget https://github.com/Gecode/gecode/archive/release-6.0.0.tar.gz
```

2.3 Building

Just go to the `src` directory and run:

```
make build
```

2.4 Testing

Unison contains a test suite with a few functions where different targets and optimization goals are exercised. To execute the tests just run:

```
make test
```

2.5 Installing

The building process generates three binaries. The installation process consists in copying the binaries into the appropriate system directory. To install the binaries under the default directory `usr/local` just run:

```
make install
```

The installation directory is specified by the Makefile variable `PREFIX`. To install the binaries under an alternative directory `$DIR` just run:

```
make install PREFIX=$DIR
```

2.6 Running

This manual uses the iterative version of the factorial function as a running example. A possible C implementation is as follows:

```
int factorial(int n) {
  int f;
  f = 1;
  while(n > 0) {
    f *= n--;
  }
  return f;
}
```

When used as a standalone tool, Unison takes as input a function in LLVM's Machine IR format (MIR). In this format, instructions of a certain processor have already been selected. The factorial function in MIR format with Hexagon V4 instructions looks as follows:

```
--- |
  ; ModuleID = (...)
...
---
name:            factorial
body:             |
  bb.0.entry (freq 12):
    liveins: %r0

    %5 = COPY %r0
    %6 = A2_tfrsi 1
    %7 = C2_cmpgti %5, 0
    J2_jumpf %7, %bb.2.while.end

  bb.1.while.body (freq 255):

    %0 = PHI %6, %bb.0.entry, %3, %bb.1.while.body
    %1 = PHI %5, %bb.0.entry, %2, %bb.1.while.body
    %2 = A2_addi %1, -1
    %3 = M2_mpyi %1, %0
    %8 = C2_cmpgti %1, 1
    J2_jumpt %8, %bb.1.while.body
    J2_jump %bb.2.while.end

  bb.2.while.end (freq 12):
    liveouts: %r0

    %4 = PHI %6, %bb.0.entry, %3, %bb.1.while.body
    %r0 = COPY %4
    JMPret %r31

...
```

To execute Unison on this function and obtain the optimal register allocation and instruction schedule for Hexagon V4, just run the following command from the top of the Git repository:

```
uni run doc/code/factorial.mir --goal=speed
```

This command outputs a function in MIR format where registers are allocated and instructions are scheduled. The function is thus already very close to assembly code:

```
--- |
  ; ModuleID = (...)

...
---
name:            factorial
body:             |
  bb.0 (freq 4):
    successors: %bb.2(1), %bb.1(1)

    BUNDLE {
      %r1 = A2_tfrsi 1
      J4_cmpgti_f_jumpnv_t %r0, 0, %bb.2, implicit %pc, implicit-def %pc
    }

  bb.1 (freq 85):
    successors: %bb.1(1), %bb.2(1)

    BUNDLE {
      %r0 = A2_addi %r0, -1
      %r1 = M2_mpyi %r0, %r1
      %p0 = C2_cmpgti %r0, 1
      J2_jumpt %p0, %bb.1, implicit %pc, implicit-def %pc
    }

  bb.2 (freq 4):

    BUNDLE {
      %r0 = A2_tfr %r1
      JMPret %r31, implicit %pc, implicit-def %pc
    }

...
```

The `uni` tool has several options and commands such as `run`. Detailed information about each option and command can be obtained by running:

```
uni --help
```

Unison can be used as a standalone tool as illustrated above but is only really useful as a complement to a full-fledged compiler. The next section gives instructions to use Unison together with LLVM.

Chapter 3

LLVM Integration

Unison is accompanied with a driver that allows transparent integration with the LLVM compiler infrastructure. In particular, the driver enables LLVM's code generator (`llc`) to run Unison transparently instead of its standard register allocation and instruction scheduling algorithms. Figure 3.1 shows how the LLVM driver interfaces with Unison to produce assembly code all the way from source code. Arcs between components are labeled with the file extension corresponding to the shared data file.

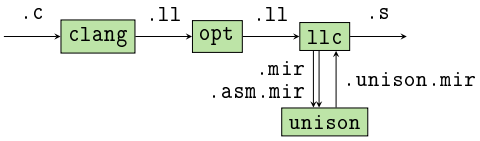

Figure 3.1: design of the LLVM driver

Unison uses LLVM's Machine IR format (MIR) as the interface language. Unison takes as input a function in MIR format (`.mir`) and the function where LLVM has already performed register allocation and instruction scheduling (`.asm.mir`) as a starting point for the optimization algorithm. For our running example, the starting solution `factorial.asm.mir` looks as follows:

```
--- |
  ; ModuleID = (...)
...
---
name:            factorial
body:             |
  bb.0 (freq 12):
    successors: %bb.2(1), %bb.1(1)

    BUNDLE {
      %r1 = A2_tfr %r0
      %r0 = A2_tfrsi 1
    }
    BUNDLE {
      %p0 = C2_cmpgti %r1, 0
      J2_jumpfnew %p0, %bb.1, implicit %pc, implicit-def %pc
    }

  bb.1 (freq 255):
    successors: %bb.1(1), %bb.2(1)

    BUNDLE {
      %r2 = A2_addi %r1, -1
      %r0 = M2_mpyi %r1, %r0
      %p0 = C2_cmpgti %r1, 1
    }
    BUNDLE {
      %r1 = A2_tfr %r2
      J2_jumpt %p0, %bb.1, implicit %pc, implicit-def %pc
    }
    J2_jump %bb.2, implicit-def %pc

  bb.2 (freq 12):

    JMPret %r31, implicit %pc, implicit-def %pc

...
```

The result of running Unison (`.unison.mir`) is sent back to `llc` where the final assembly code is emitted.

As for the core Unison tool, the driver must be built from source as we do not yet provide precompiled packages. The driver is known to work on Linux and should work in all other main platforms provided that Unison itself can be built successfully.

3.1 Downloading

The Unison driver for LLVM is hosted as a LLVM form by GitHub. The easiest way to access its source code (including the history) is by running:

```
git clone https://github.com/unison-code/llvm.git
```

LLVM 3.8 is the latest LLVM supported version. To access the driver for this version run the following command on the cloned repository:

```
git checkout release_38-unison
```

3.2 Prerequisites

The LLVM driver depends on Unison being installed successfully. Check the LLVM website for the prerequisites to build LLVM itself.

3.3 Building, Testing, and Installing

Just follow the instructions provided at LLVM's website as usual.

3.4 Running with llc

To execute `llc` such that Unison is used for register allocation and instruction scheduling, just run the following command from the top of the core Unison Git repository:

```
llc doc/code/factorial.ll -march=hexagon -mcpu=hexagonv4 -unison
```

Currently, Unison supports the following LLVM targets (defined by `march-mcpu-mattr` triples):

target	`-march=`	`-mcpu=`	`-mattr=`
Hexagon V4	`hexagon`	`hexagonv4`	
ARM1156T2F-S	`arm`	`arm1156t2f-s`	`+thumb-mode`
MIPS32	`mips`	`mips32`	

Other flags (with a `-unison` prefix) can be used to control the execution of Unison, run `llc -help` for details.

3.5 Running with clang

To execute `clang` with Unison's register allocation and instruction scheduling for Hexagon V4, first build and install a matching `clang` version (see the Clang website for details). Then just run the following command from the top of the core Unison Git repository:

```
clang doc/code/factorial.c -target hexagon -mllvm -unison
```

Alternatively, functions can be annotated with the `"unison"` attribute to indicate that Unison should be run on them:

```
__attribute__((annotate("unison")))
```

Part II

Developing and Extending Unison

Chapter 4

Architecture

As usual in compiler construction, Unison is organized as a chain of transformation components through which the program flows. Each intermediate representation of the program is stored in a file. Unison takes a mandatory and an optional file as input. The mandatory file is a function in LLVM's Machine IR format (`.mir`) where:

- instructions of a certain processor have already been selected,
- the code is in Static Single Assignment (SSA) form,
- instructions use and define temporaries (program variables at the code generation level) rather than processor registers, and
- instructions are not yet scheduled.

The optional file (`.asm.mir`) is also in Machine IR format and represents the same function where register allocation and instruction scheduling has already been applied by an external tool (typically `llc`). This file contains as a structured representation of assembly code that Unison can take as a starting solution within the optimization process. Unison generates as output a single function (`.unison.mir`) with the given function after register allocation and instruction scheduling. If Unison cannot improve the initial solution provided in the `.asm.mir` file, this is just shown as output.

Figure 4.1 shows the main components involved in compilation of intermediate (`.mir`) code to assembly (`.unison.mir`) code. Arcs between components are labeled with the file extension corresponding to the shared data file. The Unison components are enclosed by a dashed rectangle.

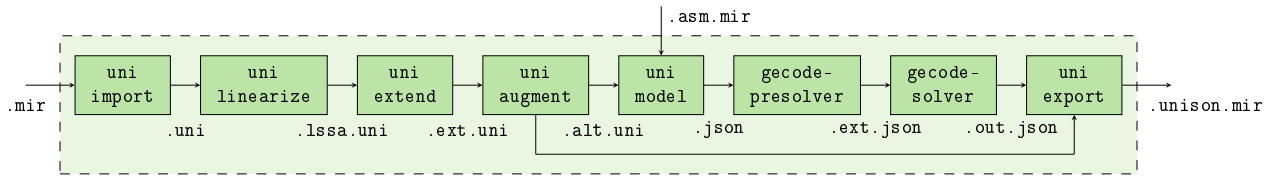

Figure 4.1: main components and boundaries of Unison

The function of each component is:

`uni import`**:** transform the instruction-selected program into Unison IR;

`uni linearize`**:** transform the program to Linear Static Single Assignment form;

`uni extend`**:** extend the program with copies;

`uni augment`**:** augment the program with alternative temporaries;

`uni model`**:** formulate a combinatorial problem combining global register allocation and instruction scheduling;

`gecode-presolver`**:** produce an equivalent combinatorial problem that is easier to solve;

`gecode-solver`**:** solve the combinatorial problem;

`uni export`**:** generate the almost-assembly program with the solution to the combinatorial problem.

Chapter 5 gives further detail on the Unison IR (`.uni`) and the different transformations that are applied to it.

Chapter 5

Unison IR

The intermediate representation (IR) used in Unison is simply referred to as *Unison IR*. Unison IR is a low-level, control-flow graph-based IR (just like LLVM's Machine IR) that exposes the structure of the program and the multiple register allocation and instruction scheduling decisions to be formulated in the combinatorial model. Unison IR has the following distinguishing features:

linear static single assignment form (LSSA) LSSA is a program form in which temporaries (program variables at Unison IR's level) are local to basic blocks (*blocks* for short) and relations across temporaries from different blocks are made explicit.

optional copies Unison IR includes optional copy operations that can be deactivated or implemented by alternative instructions.

alternative temporaries Unison IR allows operations to use alternative temporaries that hold the same value.

Unison IR as required for the combinatorial model formulation is constructed incrementally in four transformations as shown in Figure 4.1. This section introduces the elements of Unison IR progressively by following the transformation chain for the running example.

5.1 Initial Unison IR

The initial Unison IR (`.uni`) after running the `uni import` component has a very similar structure to the input MIR function (`.mir`). This is how `factorial.uni` looks like:

```
function: factorial
b0 (entry, freq: 4):
    o0: [t0:r0,t1:r31] <- (in) []
    o1: [t2] <- A2_tfrsi [1]
    o2: [t3] <- C2_cmpgti [t0,0]
    o3: [] <- J2_jumpf [t3,b2]
    o4: [] <- (out) []
b1 (freq: 85):
    o5: [] <- (in) []
    o6: [t4] <- (phi) [t2,b0,t7,b1]
    o7: [t5] <- (phi) [t0,b0,t6,b1]
    o8: [t6] <- A2_addi [t5,-1]
    o9: [t7] <- M2_mpyi [t5,t4]
    o10: [t8] <- C2_cmpgti [t5,1]
    o11: [] <- J2_jumpt [t8,b1]
    o12: [] <- (out) []
b2 (exit, return, freq: 4):
    o13: [] <- (in) []
    o14: [t9] <- (phi) [t2,b0,t7,b1]
    o15: [] <- JMPret [t1]
    o16: [] <- (out) [t9:r0]
adjacent:
rematerializable:
fixed-frame:
frame:
stack-pointer-offset: 0
stack-arg-size: 0
jump-table:
goal: speed
removed-freqs:
source:
; ModuleID = (...)
```

A Unison IR file consists of multiple sections:

`function`**:** name of the function.

code: actual code of the function (more details are given below).

`adjacent`**:** relations among temporaries from different blocks (more details are given in Section 5.2).

`fixed-frame`**:** frame objects (such as arguments passed by the stack) with a fixed position.

`frame`**:** frame objects with a variable position.

`stack-pointer-offset`**:** offset to add to the positions of the frame objects in the stack frame.

`jump-table`: blocks to which jump tables in the code can jump.

`goal`: optimization goal for Unison (either `speed` or `size`).

`source`: source code from which the function originates (typically LLVM IR).

The code section consists of a list of blocks. Each block has some attributes (whether it is the entry or exit point of the function, whether it returns to its caller function, and its estimated or profiled execution frequency) and a list of operations. An operation consists of an identifier (for example `o8`), a list of definition operands (`[t6]`), an instruction that implements the operation (`A2_addi`), and a list of use operands (`[t5, -1]`). Definition operands are always temporaries (`t6`); use operands can be temporaries (`t5`), block references (`b2` in `o3`, or Machine IR operands (`-1` in `o8`) that are only passed through by Unison.

Some Unison IR operations are virtual, that is, they contribute to the definition of the function semantics but do not appear in Unison's output. An example is the `in` and `out` delimiter operations, which define (use) temporaries that are live-in (out) at the entry (exit) point of each block. Another example is the `phi` operations in blocks `b1` and `b2` (the initial Unison IR is in Static Single Assignment (SSA) form). A `phi` operation (for example `o14`) at the beginning of a block defines a temporary with the value of different temporaries depending on the preceding block that is executed (for example `t9` is set to the value of `t2` if `b0` is the preceding block or `t7` if `b1` is the preceding block).

In the Unison IR, a preassignment of a temporary `t` to a register `r` at a certain point is represented by `t:r`. For example, the calling convention in Hexagon dictates that the first argument of a function is passed in register `r0`, which is modeled by the definition operands `t0:r0` in the `(in)` delimiter of the entry block `b0`.

5.2 Linearized Unison IR

The linearized Unison IR (`.lssa.uni`) after running the `uni linearize` component is in Linear Static Single Assignment (LSSA) form. This is how `factorial.lssa.uni` looks like:

```
function: factorial
b0 (entry, freq: 4):
    o0: [t0:r0,t1:r31] <- (in) []
    o1: [t2] <- A2_tfrsi [1]
    o2: [t3] <- C2_cmpgti [t0,0]
    o3: [] <- J2_jumpf [t3,b2]
    o4: [] <- (out) [t0,t1,t2]
b1 (freq: 85):
    o5: [t4,t5,t6] <- (in) []
    o6: [t7] <- A2_addi [t5,-1]
    o7: [t8] <- M2_mpyi [t5,t4]
    o8: [t9] <- C2_cmpgti [t5,1]
    o9: [] <- J2_jumpt [t9,b1]
    o10: [] <- (out) [t6,t7,t8]
b2 (exit, return, freq: 4):
    o11: [t10,t11] <- (in) []
    o12: [] <- JMPret [t11]
    o13: [] <- (out) [t10:r0]
adjacent:
    t0 -> t5, t1 -> t6, t1 -> t11, t2 -> t4, t2 -> t10, t6 -> t6, t6 -> t11,
    t7 -> t5, t8 -> t4, t8 -> t10
rematerializable:
fixed-frame:
frame:
stack-pointer-offset: 0
stack-arg-size: 0
jump-table:
goal: speed
removed-freqs:
source:
; ModuleID = (...)
```

The difference with the initial Unison IR is that each temporary `t` (for example `t1` in `factorial.uni`) is decomposed into one temporary per block where `t` is live (for example `t1` in `b0`, `t6` in `b1`, and `t11` in `b2`). The congruence among LSSA temporaries that originate from the same temporary is kept in the `adjacent` section. The fact that a temporary `t` in a block is congruent to another temporary `t'` in an immediate successor block is represented by `t -> t'` (for example `t1 -> t6` indicates that `t1` in `b0` is congruent to `t6` in the immediate successor block `b1`).

In the linearized Unison IR, liveness information is made explicit by defining and using the decomposed temporaries in `(out)` and `(in)` delimiter operations. In this form, `(phi)` operations are no longer necessary as the information that they convey is captured by the new elements in the IR. For example, the relation between `t9`, `t2`, and `t7` in `factorial.uni` is captured by `t2 -> t10` and `t8 -> t10` in `factorial.lssa.uni` (`t2`, `t7`, and `t9` in `factorial.uni` correspond to `t2`, `t8`, and `t10` in `factorial.lssa.uni`).

5.3 Extended Unison IR

The transformation performed by `uni extend` (`.ext.uni`) extends the IR with optional copy operations that can be deactivated by the solver or used to support register allocation decisions such as spilling and live-range splitting. This is how `factorial.ext.uni` looks like:

```
function: factorial
b0 (entry, freq: 4):
    o0: [t0:r0,t1:r31] <- (in) []
    o1: [t2] <- {-, MVW, STW} [t0]
    o2: [t3] <- A2_tfrsi [1]
    o3: [t4] <- {-, MVW, STW, STW_nv} [t3]
    o4: [t5] <- {-, MVW, LDW} [t2]
    o5: [t6] <- C2_cmpgti [t5,0]
    o6: [t7] <- {-, MVW, LDW} [t2]
    o7: [t8] <- {-, MVW, LDW} [t4]
    o8: [] <- J2_jumpf [t6,b2]
    o9: [] <- (out) [t7,t1,t8]
b1 (freq: 85):
    o10: [t9,t10,t11] <- (in) []
    o11: [t12] <- {-, MVW, STW} [t9]
    o12: [t13] <- {-, MVW, STW} [t10]
    o13: [t14] <- {-, MVW, LDW} [t13]
    o14: [t15] <- A2_addi [t14,-1]
    o15: [t16] <- {-, MVW, STW, STW_nv} [t15]
    o16: [t17] <- {-, MVW, LDW} [t12]
    o17: [t18] <- {-, MVW, LDW} [t13]
    o18: [t19] <- M2_mpyi [t18,t17]
    o19: [t20] <- {-, MVW, STW, STW_nv} [t19]
    o20: [t21] <- {-, MVW, LDW} [t13]
    o21: [t22] <- C2_cmpgti [t21,1]
    o22: [t23] <- {-, MVW, LDW} [t16]
    o23: [t24] <- {-, MVW, LDW} [t20]
    o24: [] <- J2_jumpt [t22,b1]
    o25: [] <- (out) [t11,t23,t24]
b2 (exit, return, freq: 4):
    o26: [t25,t26] <- (in) []
    o27: [t27] <- {-, MVW, STW} [t25]
    o28: [t28] <- {-, MVW, LDW} [t27]
    o29: [] <- JMPret [t26]
    o30: [] <- (out) [t28:r0]
adjacent:
    t1 -> t11, t1 -> t26, t7 -> t10, t8 -> t9, t8 -> t25, t11 -> t11,
    t11 -> t26, t23 -> t10, t24 -> t9, t24 -> t25
rematerializable:
fixed-frame:
frame:
stack-pointer-offset: 0
stack-arg-size: 0
jump-table:
goal: speed
removed-freqs:
source:
; ModuleID = (...)
```

A copy operation can be discarded or implemented by alternative instructions. For example,

`o15` can be deactivated if implemented by the special *null* instruction (`-`), or implemented by a register-to-register move (`MVW`), a regular store (`STW`), or a zero-read-latency store (`STW_nv`). The particular strategy to extend functions with copies is processor-specific (see Chapter 7), but it is common to insert a copy including store instructions after each temporary definition and a copy including load instructions after each temporary use. For example, `o15` and `o22` are inserted after the definition and before a use of `t25` and `t23` which correspond to `t7` in `factorial.lssa.uni`.

5.4 Augmented Unison IR

The augmented IR (`.alt.uni`) allows operations to use alternative temporaries that hold the same value. This is how factorial.alt.uni looks like:

```
function: factorial
b0 (entry, freq: 4):
    o0: [p0{t0}:r0,p1{t1}:r31] <- (in) []
    o1: [p3{-, t2}] <- {-, MVW, STW} [p2{-, t0}] (reads: [control])
    o2: [p4{t3}] <- A2_tfrsi [1] (reads: [control])
    o3: [p6{-, t4}] <- {-, MVW, STW, STW_nv} [p5{-, t3}] (reads: [control])
    o4: [p8{-, t5}] <- {-, MVW, LDW} [p7{-, t0, t2}] (reads: [control])
    o5: [p10{-, t6}] <- {-, C2_cmpgti} [p9{-, t0, t2, t5, t7},0] (reads: [control])
    o6: [p12{-, t7}] <- {-, MVW, LDW} [p11{-, t0, t2}] (reads: [control])
    o7: [p14{-, t8}] <- {-, MVW, LDW} [p13{-, t3, t4}] (reads: [control])
    o8: [p16{-, t9}] <- {-, J2_jumpf_linear, J2_jumpf_nv_linear} [p15{-, t6}] (writes: [control])
    o9: [p18{-, t10}] <- {-, J4_cmpgti_f_jumpnv_t_linear} [p17{-, t0, t2, t5, t7},0] (writes: [control])
    o10: [] <- jump_merge [p19{t9, t10},b2] (writes: [control,pc])
    o11: [] <- (out) [p20{t0, t2, t5, t7},p21{t1},p22{t3, t4, t8}]
b1 (freq: 85):
    o12: [p23{t11},p24{t12},p25{t13}] <- (in) []
    o13: [p27{-, t14}] <- {-, MVW, STW} [p26{-, t11}] (reads: [control])
    o14: [p29{-, t15}] <- {-, MVW, STW} [p28{-, t12}] (reads: [control])
    o15: [p31{-, t16}] <- {-, MVW, LDW} [p30{-, t12, t15}] (reads: [control])
    o16: [p33{t17}] <- A2_addi [p32{t12, t15, t16, t20, t23},-1] (reads: [control])
    o17: [p35{-, t18}] <- {-, MVW, STW, STW_nv} [p34{-, t17}] (reads: [control])
    o18: [p37{-, t19}] <- {-, MVW, LDW} [p36{-, t11, t14}] (reads: [control])
    o19: [p39{-, t20}] <- {-, MVW, LDW} [p38{-, t12, t15}] (reads: [control])
    o20: [p42{t21}] <- M2_mpyi [p40{t12, t15, t16, t20, t23},p41{t11, t14, t19}] (reads: [control])
    o21: [p44{-, t22}] <- {-, MVW, STW, STW_nv} [p43{-, t21}] (reads: [control])
    o22: [p46{-, t23}] <- {-, MVW, LDW} [p45{-, t12, t15}] (reads: [control])
    o23: [p48{-, t24}] <- {-, C2_cmpgti} [p47{-, t12, t15, t16, t20, t23},1] (reads: [control])
    o24: [p50{-, t25}] <- {-, MVW, LDW} [p49{-, t17, t18}] (reads: [control])
    o25: [p52{-, t26}] <- {-, MVW, LDW} [p51{-, t21, t22}] (reads: [control])
    o26: [p54{-, t27}] <- {-, J2_jumpt_linear, J2_jumpt_nv_linear} [p53{-, t24}] (writes: [control])
    o27: [p56{-, t28}] <- {-, J4_cmpgti_t_jumpnv_t_linear} [p55{-, t12, t15, t16, t20, t23},1] (writes: [control])
    o28: [] <- jump_merge [p57{t27, t28},b1] (writes: [control,pc])
    o29: [] <- (out) [p58{t13},p59{t17, t18, t25},p60{t21, t22, t26}]
b2 (exit, return, freq: 4):
    o30: [p61{t29},p62{t30}] <- (in) []
    o31: [p64{-, t31}] <- {-, MVW, STW} [p63{-, t29}] (reads: [control])
    o32: [p66{-, t32}] <- {-, MVW, LDW} [p65{-, t29, t31}] (reads: [control])
    o33: [] <- JMPret [p67{t30}] (reads: [r31], writes: [control,pc,pc])
    o34: [] <- (out) [p68{t29, t31, t32}:r0]
adjacent:
    p20 -> p24, p21 -> p25, p21 -> p62, p22 -> p23, p22 -> p61, p58 -> p25,
    p58 -> p62, p59 -> p24, p60 -> p23, p60 -> p61
rematerializable:
fixed-frame:
frame:
stack-pointer-offset: 0
stack-arg-size: 0
jump-table:
goal: speed
removed-freqs:
source:
; ModuleID = (...)
```

The main change in the augmented Unison IR is the introduction of operand identifiers (for

example p47) and temporaries that can be *connected* to them (for example {-, t12, t15, ...}) where the special *null* connection (-) indicates that the operand is not connected to any temporary because its operation is inactive.

Another difference is the annotation of operations with side effects. A side effect reads or writes an abstract object (for example, control for control flow or pc for Hexagon's program counter). Multiple reads and writes to the same object cause dependencies among the operations (for example, o2 must be issued before o8 provided the latter is active as o2 reads and o8 writes the control object.

Finally, a Hexagon specific transformation is performed where an alternative way of implementing compare-and-jump operations (o5 and o8; o21 and o24) is introduced (see the Hexagon Programmer's Reference Manual or the comments in LLVM's HexagonNewValueJump.cpp file for further detail).

Chapter 6

Combinatorial Model

This chapter formulates the combinatorial model of register allocation and instruction scheduling that is at the core of Unison. The combinatorial model consists of parameters (Section 6.1) describing the input program, processor, and objective; variables (Section 6.2) capturing the different decisions involved in register allocation and instruction scheduling; constraints (Section 6.3) relating and limiting the decisions; and an objective function (Section 6.4) to optimize for. This chapter provides a raw but formal description of the model, for further explanations please consult [1].

6.1 Parameters

This section lists the parameters of the combinatorial model with examples from `factorial.json` (see Figure 4.1).

6.1.1 Program

B, O, P, T	sets of blocks, operations, operands and temporaries
`B`	`[0, 1, 2]`
`O`	`[0, 1, 2, ..., 33, 34]`
`P`	`[0, 1, 2, ..., 67, 68]`
`T`	`[0, 1, 2, ..., 31, 32]`
block(o)	block to which operation o belongs
`block`	`[0, 0, ..., 1, 1, 1, 2, 2, 2, 2, 2]`
operands(o)	set of operands of operation o
`operands`	`[[0, 1], [2, 3], [4], ..., [65, 66], [67], [68]]`
temps(p)	set of temporaries that can be connected to operand p
`temps`	`[[0], [1], [-1, 0], ..., [30], [29, 31, 32]]`

use(p)	whether p is a use operand
`use`	`[false, false, true, ..., true, true]`
$p \xrightarrow{=} q$	whether operands p and q are adjacent
`adjacent`	`[[20, 24], [21, 25], ..., [60, 61]]`
$p \triangleright$`r`	whether operand p is preassigned to register `r`
`preassign`	`[[0, 0], [1, 31], [68, 0]]`
width(t)	number of register atoms that temporary t occupies
`width`	`[1, 1, 1, ..., 1, 1]`
freq(b)	estimated execution frequency of block b
`freq`	`[4, 85, 4]`
min-live(t)	minimum live duration of temporary t if it is live
`minlive`	`[1, 1, 1, ..., 1, 1]`
dep(b)	fixed dependency graph of the operations of block b
`dep`	`[[[0, 1], ..., [1, 8], ..., [10, 11]], [...], [...]]`
prescheduled(o, `c`)	whether operation o is *prescheduled* to cycle `c`
`preschedule`	`[[2, 3], [1, 1], [60, 24]]` *note: the example JSON array is extracted from a different program (Hexagon programs do not yet yield prescheduling constraints)*
out(b)	out-delimiter of block b
`out`	`[11, 29, 34]`

6.1.2 Processor

I, R	sets of instructions and resources
`I`	`[0, 1, 2, 3, ..., 16]`
`R`	`[0, 1, 2, 3, ..., 8]`
dist(o_1, o_2, `i`)	min. issue distance of ops. o_1 and o_2 when o_1 is implemented by `i`
`dist`	`[[[1], ..., [0, 0, 0], ..., [1]], [...], [...]]` *note: this parameter is encoded with the same structure as* `dep`*: each dependency and its corresponding distance array are found in the same positions of their respective JSON arrays (example:* `dep[0][2] = [0, 3]`*,* `dist[0][2] = [1]`*).*
class(o, `i`, p)	register class in which operation o implemented by `i` accesses p
`class`	`[[0, 0]], [[0, 0], [1, 1], [1, 9]], ..., [[0]]`
atoms($r\,c$)	atoms of register class $r\,c$
`atoms`	`[[0, 1, 2, ..., 76], ..., [37, 39, 41, ..., 75]]`

instrs(o)	set of instructions that can implement operation o
`instructions`	`[[2], [0, 3, 4], [5], ..., [16], [2]]`
lat(o,`i`,p)	latency of p when its operation o is implemented by `i`
`lat`	`[[[1, 1]], [[0, 0], [0, 1], [0, 1]], ..., [[0]]]`
bypass(o,`i`,p)	whether p is bypassing when its operation o is implemented by `i`
`bypass`	`[[[false, false]], [[false, false], ..., [[false]]]`
cap(`r`)	capacity of processor resource r
`cap`	`[4, 4, 2, 1, 2, 1, 1, 2, 1]`
con(`i`,`r`)	consumption of processor resource r by instruction `i`
`con`	`[[0, 0, ..., 0, 0], ..., [1, 1, 0, 0, 1, 1, 0, 0, 0]]`
dur(`i`,`r`)	duration of usage of processor resource r by instruction `i`
`dur`	`[[0, 0, ..., 0, 0], ..., [1, 1, 0, 0, 1, 1, 0, 0, 0]]`
off(`i`,`r`)	offset of usage of processor resource r by instruction `i`
`off`	`[[0, 0, ..., 0, 0], ..., [0, 0, 0, 0, 0, 0, 0, 0, 0]]`
aligned(p,i,q,j)	whether operands p and q are aligned when implemented by instructions i and j
`aligned`	`[[69, 17, 71, 17], [70, 24, 71, 24]]` *note: the example JSON arrays are extracted from a different program (Hexagon programs do not yield alignment constraints)*
adist(p,i,q,j)	alignment distance of operands p and q when implemented by instructions i and j
`adist`	`[0, 1, 1]` *note: this parameter is encoded with the same structure as* `aligned`*: each aligned operand tuple and its corresponding alignment distance are found in the same positions of their respective JSON arrays (example:* `aligned[1] = [70, 24, 71, 24]`, `adist[1] = 1`*.)*
packed(p,q)	whether operands p and q are packed
`packed`	`[[13, 14], [34, 35], [54, 55]]` *note: the example JSON arrays are extracted from a different program (Hexagon programs do not yield packing constraints)*
exrelated(p,q)	whether operands p and q are related extensionally
`exrelated`	`[[4, 5]]` *note: the example JSON arrays are extracted from a different program (Hexagon programs do not yield extensional constraints)*
table(p,q)	table of register assignments for operands p and q
`table`	`[[0, 1],[2, 3],[4, 5],[6, 7]]`

note: the example JSON arrays are extracted from a different program (Hexagon programs do not yield extensional constraints)

activators(o)
: set of instructions that activate operation o

`activators`
: `[[], [10, 17, 13, 19], [10, 17, 13, 19], ..., []]`
note: the example JSON array is extracted from a different program (Hexagon programs do not yet yield activation constraints)

CS
: caller-saved atoms

`callersaved`
: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 28, 32, 33, 34, 35]`

E
: set of ad hoc processor constraints

`E`
: `[[2, [1, [5, 13], [5, 14]], [5, 15]], ...]`
note: constraints are encoded as trees of expression tuples, where the first element of each expression tuple encodes its type as follows:

0	or	8	distance
1	and	9	share
2	xor	10	operand-overlap
3	implies	11	temporary-overlap
4	not	12	caller-saved
5	active	13	allocated
6	connects	14	aligned
7	implements		

note: the example JSON array is extracted from a different program (Hexagon programs do not yet yield ad hoc constraints)

6.1.3 Objective

dynamic (n)
: whether to use block frequencies as weight for the n^{th} objective

`optimize_dynamic`
: `[true]`

resource (n)
: resource whose consumption is to be optimized for the n^{th} objective

`optimize_resource`
: `[-1]`
note: the estimated number of cycles (`cycles`*) is encoded as resource* `-1` *, otherwise the usual resource numbers in* `R` *are used*

max-cost (n)
: upper bound of the n^{th} objective

`maxf`
: `[274]`

6.2 Variables

$a_o \in \{0,1\}$	whether operation o is active
$i_o \in \text{instrs}(o)$	instruction that implements operation o
$l_t \in \{0,1\}$	whether temporary t is live
$r_t \in \mathbb{N}_0$	register to which temporary t is assigned
$x_p \in \{0,1\}$	whether operand p is connected
$y_p \in \text{temps}(p)$	temporary that is connected to operand p
$c_o \in \mathbb{N}_0$	issue cycle of operation o relative to the beginning of its block
$ls_t \in \mathbb{N}_0$	live start of temporary t
$le_t \in \mathbb{N}_0$	live end of temporary t
$s_p \in \mathbb{Z}$	latency slack of global operand p

6.3 Constraints

6.3.1 Register Allocation

connected operands: operands cannot be connected to null temporaries.

$$x_p \iff y_p \neq \bot \quad \forall p \in P \tag{6.1}$$

$$\textit{example:} \quad x_{p_5} \iff y_{p_5} \neq \bot$$

connected users: a temporary is live iff it is connected to a user.

$$l_t \iff \exists p \in \text{users}(t) : y_p = t \quad \forall t \in T \tag{6.2}$$

where

$$\text{users}(t) = \left\{ p \in P : \text{use}(p) \wedge t \in \text{temps}(p) \right\} \tag{6.3}$$

$$\textit{example:} \quad l_{t_4} \iff \exists p \in \left\{ p_{13}, p_{22} \right\} : y_p = t_4$$

connected definers: a temporary is live iff it is connected to its definer.

$$l_t \iff x_{\text{definer}(t)} \quad \forall t \in T \tag{6.4}$$

where

$$\text{definer}(t) = p \in P : \left(\neg \text{use}(p) \wedge t \in \text{temps}(p) \right) \tag{6.5}$$

$$\textit{example:} \quad l_{t_{19}} \iff x_{p_{37}}$$

local operand connections: local operands are connected iff their operations are active.

$$x_p \iff a_{\text{operation}(p)} \quad \forall p \in P : \neg \text{global}(p) \tag{6.6}$$

where

$$\text{global}(p) \iff \exists q \in P : \left(p \xrightarrow{=} q \vee q \xrightarrow{=} p\right) \quad \forall p \in P \tag{6.7}$$

and

$$\text{operation}(p) = o \in O : p \in \text{operands}(o) \tag{6.8}$$

$$\textit{example:} \quad x_{p_{45}} \iff a_{o_{22}}$$

global operand connections: global operands are connected iff any of their successors is connected.

$$x_p \iff \exists q \in P : \left(p \xrightarrow{=} q \wedge x_q\right) \quad \forall p \in P : \text{global}(p) \tag{6.9}$$

$$\textit{example:} \quad x_{p_{21}} \iff \exists q \in \left\{p_{25}, p_{62}\right\} : x_q$$

active instructions: active operations are implemented by non-null instructions.

$$a_o \iff i_o \neq \perp \quad \forall o \in O \tag{6.10}$$

$$\textit{example:} \quad a_{o_{15}} \iff i_{o_{15}} \neq \perp$$

register class: the instruction that implements an operation determines the register class to which its operands are allocated.

$$r_{y_p} \in \text{atoms}(\text{class}(o, i_o, p)) \quad \forall o \in O, \forall p \in \text{operands}(o) \tag{6.11}$$

$$\textit{example:} \quad r_{y_{p_{14}}} \in \text{class}(o_7, i_{o_7}, p_{14})$$

disjoint live ranges: temporaries whose live ranges overlap are assigned to different register atoms.

$$\text{disjoint2}(\{\langle r_t, r_t + \text{width}(t) \times l_t, ls_t, le_t \rangle : t \in \text{T}(b)\}) \ \forall b \in B \tag{6.12}$$

where

$$\text{T}(b) = \{t \in T : \text{block}(\text{operation}(\text{definer}(t))) = b\} \tag{6.13}$$

$$\textit{example:} \quad \text{disjoint2}(\{\left\langle r_{t_0}, r_{t_0} + 1 \times l_{t_0}, ls_{t_0}, le_{t_0} \right\rangle, \ldots, \left\langle r_{t_{10}}, r_{t_{10}} + 1 \times l_{t_{10}}, ls_{t_{10}}, le_{t_{10}} \right\rangle\})$$

preassignment: certain operands are preassigned to registers.

$$r_{y_p} = \texttt{r} \quad \forall p \in P : p \triangleright \texttt{r} \tag{6.14}$$

$$\textit{example:} \quad r_{y_{p_1}} = \texttt{r31}$$

congruence: connected adjacent operands are assigned to the same register.

$$x_p \wedge x_q \implies r_{y_p} = r_{y_q} \quad \forall p, q \in P : p \xrightarrow{=} q \tag{6.15}$$

$$\textit{example:} \quad x_{p_{21}} \wedge x_{p_{25}} \implies r_{y_{p_{21}}} = r_{y_{p_{25}}}$$

alignment: aligned operands are assigned to registers at a given relative distance.

$$\begin{aligned} i_{\text{operation}(p)} = i \wedge i_{\text{operation}(q)} = j \implies r_{y_p} = r_{y_q} + \text{adist}(p,i,q,j) \\ \forall p,q \in P, \forall i,j \in I : \text{aligned}(p,i,q,j) \end{aligned} \tag{6.16}$$

packing: packed operands are assigned to contiguous, complementary registers.

$$\begin{aligned} x_p \wedge x_q \implies r_{y_q} = r_{y_p} + \begin{cases} \text{width}(p) & \text{if } r_{y_p} \bmod \big(\text{width}(p) \times 2\big) = 0 \\ -\text{width}(p) & \text{otherwise} \end{cases} \\ \forall p,q \in P : \text{packed}(p,q) \end{aligned} \tag{6.17}$$

extensional: the registers assigned to some pairs of operands are related extensionally.

$$\text{extensional}(\langle p,q \rangle, \text{table}(p,q)) \quad \forall p,q \in P : \text{exrelated}(p,q) \tag{6.18}$$

6.3.2 Instruction Scheduling

live start: the live range of a temporary starts at the issue cycle of its definer.

$$l_t \implies ls_t = c_{\text{operation}(\text{definer}(t))} \quad \forall t \in T \tag{6.19}$$

live end: the live range of a temporary ends with the last issue cycle of its users.

$$l_t \implies le_t = \max \left(\max_{\substack{p \in \text{users}(t): \\ y_p = t}} c_{\text{operation}(p)}, ls_t + \text{min-live}(t) \right) \quad \forall t \in T \tag{6.20}$$

data precedences: an operation that uses a temporary must be preceded by its definer.

$$\begin{aligned} y_q = t \implies c_u \geq c_d + \text{lat}(d,i_d,p) + \text{slack}(p) + \text{lat}(u,i_u,q) + \text{slack}(q) \\ \forall t \in T, \\ \forall p \in \{\text{definer}(t)\}, \forall d \in \big\{\text{operation}(p)\big\} \\ \forall q \in \text{users}(t), \forall u \in \big\{\text{operation}(q)\big\} \end{aligned} \tag{6.21}$$

where

$$\text{slack}(p) = \begin{cases} s_p & \text{if global}(p) \\ 0 & \text{otherwise} \end{cases} \tag{6.22}$$

processor resources: the capacity of each processor resource cannot be exceeded at any issue cycle.

$$\begin{aligned} \text{cumulative}(\{\langle c_o + \text{off}(i_o,\mathtt{r}), \text{con}(i_o,\mathtt{r}), \text{dur}(i_o,\mathtt{r}) \rangle : o \in \text{O}(b)\}, \text{cap}(\mathtt{r})) \\ \forall b \in B, \forall \mathtt{r} \in R \end{aligned} \tag{6.23}$$

where

$$\mathrm{O}(b) = \{o \in O : \mathrm{block}(o) = b\} \tag{6.24}$$

fixed precedences: control and read-write dependencies yield fixed precedences among operations.

$$a_d \wedge a_u \implies c_u \geq c_d + \mathrm{dist}(d, u, i_d) \quad \forall b \in B, \forall (d,u) \in \mathrm{dep}(b) \tag{6.25}$$

activation: an operation is active if any of its activator instructions is selected.

$$\exists o' \in O : i_{o'} \in \mathrm{activators}(o) \implies a_o \quad \forall o \in O \tag{6.26}$$

slack balancing: the slack of adjacent operands is balanced.

$$s_p + s_q = 0 \quad \forall p, q \in P : p \xrightarrow{=} q \tag{6.27}$$

prescheduling: certain operations are prescheduled before the last operation issue.

$$a_o \implies c_o = \mathtt{c} \quad \forall o \in O : \mathrm{prescheduled}(o, \mathtt{c}) \tag{6.28}$$

and

$$a_o \iff \mathtt{c} < c_{\mathrm{out}(\mathrm{block}(o))} \quad \forall o \in O : \mathrm{prescheduled}(o, \mathtt{c}) \tag{6.29}$$

bypassing: the operation of a bypassing operand is scheduled in parallel with its definer.

$$\mathrm{bypass}(o, i_o, p) \implies c_o = c_{\mathrm{operation}(\mathrm{definer}(y_p))} \quad \forall o \in O, \forall p \in \mathrm{operands}(o) \tag{6.30}$$

adhoc: ad hoc processor constraints are satisfied.

$$\mathrm{satisfy}(e) \quad \forall e \in E \tag{6.31}$$

where

$$\begin{aligned}
\text{satisfy}(\langle \texttt{or}, e_1, e_2, \ldots, e_n \rangle) &\iff \text{satisfy}(e_1) \vee \text{satisfy}(e_2) \vee \ldots \vee \text{satisfy}(e_n) \\
\text{satisfy}(\langle \texttt{and}, e_1, e_2, \ldots, e_n \rangle) &\iff \text{satisfy}(e_1) \wedge \text{satisfy}(e_2) \wedge \ldots \wedge \text{satisfy}(e_n) \\
\text{satisfy}(\langle \texttt{xor}, e_1, e_2 \rangle) &\iff \text{satisfy}(e_1) \oplus \text{satisfy}(e_2) \\
\text{satisfy}(\langle \texttt{implies}, e_1, e_2 \rangle) &\iff \text{satisfy}(e_1) \implies \text{satisfy}(e_2) \\
\text{satisfy}(\langle \texttt{not}, e \rangle) &\iff \neg \text{satisfy}(e) \\
\text{satisfy}(\langle \texttt{active}, o \rangle) &\iff a_o \\
\text{satisfy}(\langle \texttt{connects}, p, t \rangle) &\iff y_p = t \\
\text{satisfy}(\langle \texttt{implements}, o, \texttt{i} \rangle) &\iff i_o = \texttt{i} \\
\text{satisfy}(\langle \texttt{distance}, d, u, n \rangle) &\iff c_u \geq c_d + n \\
\text{satisfy}(\langle \texttt{share}, p, q \rangle) &\iff y_p = y_q \\
\text{satisfy}(\langle \texttt{operand-overlap}, p, q \rangle) &\iff ls_{y_p} < le_{y_q} \wedge ls_{y_q} < le_{y_p} \\
\text{satisfy}(\langle \texttt{temporary-overlap}, t, t' \rangle) &\iff ls_t < le_{t'} \wedge ls_{t'} < le_t \\
\text{satisfy}(\langle \texttt{caller-saved}, t \rangle) &\iff r_t \in \text{CS} \\
\text{satisfy}(\langle \texttt{allocated}, p, r\,c \rangle) &\iff r_{y_p} \in \text{atoms}(r\,c) \\
\text{satisfy}(\langle \texttt{aligned}, p, q, n \rangle) &\iff r_{y_q} = r_{y_p} + n
\end{aligned}$$

6.4 Objective

The objective is to minimize the sum of the weighted costs of each block according to the first objective (multi-objective optimization is not yet supported):

$$\sum_{b \in B} \text{weight}(b) \times \text{cost}(b)$$

where $\text{weight}(b)$ gives the weight of block b:

$$\text{weight}(b) = \begin{cases} \text{freq}(b) & \text{if dynamic}\,(0) \\ 1 & \text{otherwise} \end{cases} \tag{6.32}$$

and $\text{cost}(b)$ gives the estimated cost of block b:

$$\text{cost}(b) = \begin{cases} c_{\text{out}(b)} & \text{if resource}\,(0) = \texttt{cycles} \\ \sum_{o \in \text{O}(b)} \text{con}(i_o, \text{resource}\,(0)) & \text{otherwise} \end{cases} \tag{6.33}$$

To optimize for speed, dynamic (0) is set to `true` and resource (0) is set to the special resource `cycles`. To optimize for code size, dynamic (0) is set to `false` and resource (0) is set to the processor resource `bits` representing the bits with which instructions are encoded.

Chapter 7

Target Description

This chapter explains how target processors are described in Unison. Section 7.1 describes the structure of the source code that implements a target processor description. Section 7.5 describes how major parts of the Unison description can be generated from a higher-level description. Section 7.6 explains how the higher-level description itself can be imported from LLVM.

7.1 Structure

The source code describing a new target is located under the directory `src/Unison/Target` within the base directory of the `Unison` Haskell package (`src/unison`). At the highest level, a processor description is a function that returns a parameterized data structure of type `TargetDescription i r rc s`, where `i`, `r`, `rc`, and `s` are the processor's instruction, register, register class, and resource types. They are typically defined as enumeration types, but can consist of more complex types if required by the target description. A processor is registered in Unison by simply adding the target description to a list provided by the function `unisonTargets` in the module `Unison.Target`.

The `TargetDescription` data structure consists of a collection of functions that Unison uses to query about the properties of the processor. Detailed documentation about the processor functions can be obtained by building the code documentation for the Unison package. To do this, just go to the `src` directory and run:

```
make doc
```

The generated HTML documentation can be found under the `.stack-work/dist` directory within the Unison package. The `TargetDescription` data type can be found in the module `Unison/Target/API`.

Unison includes a minimal, compilable target (`Minimal`) that can be used as a template to create new targets, see the module `Unison.Target.Minimal` in the Unison package. A simple MIR version of our running example with Minimal instructions is included in `doc/code/factorial.mir`.

To execute Unison on this function and obtain the optimal register allocation and instruction schedule for Minimal, run the following command from the top of the Git repository:

```
uni run doc/code/minimal.mir --goal=speed --target=Minimal
```

By default, Minimal is defined as a single-issue processor, but an arbitrary issue width `N` can be specified using the following target option:

```
uni run doc/code/minimal.mir ... --targetoption=issue-width:N
```

The rest of this section explains the key abstractions used in the processor description and how different parts of it can be generated automatically.

7.2 Register Array

7.3 Resource Model

7.4 Calling Conventions

7.5 Target Generation

Large parts of a Unison processor description can be generated from a high-level, YAML-based description of the instruction set. The description consists of a list of instructions, where the main attributes of each instruction (its identifier, operands, size, ...) are defined. This is how the description for Hexagon's instruction `A2_addi` in the running example looks like:

```
- id:                  A2_addi
  type:                linear
  operands:
   - Rd:               [register, def, IntRegs]
   - Rs:               [register, use, IntRegs]
   - s16:              bound
  uses:                [Rs, s16]
  defines:             [Rd]
  size:                4
  affects:
  affected-by:
  itinerary:           ALU32_ADDI_tc_1_SLOT0123
```

The high-level YAML-based descriptions are transformed by Unison's tool `specsgen` into actual Haskell code to be compiled together with the rest of the Unison project. To build `specsgen`, go to the `src` directory and run:

```
make build-specsgen
```

To install it, just run:

```
make install-specsgen
```

`specsgen` takes as input a number of YAML files (`.yaml`) and generates a number of Haskell files (`.hs`) in a given directory. Run `specsgen -help` for more details. Makefile recipes are defined to run `specsgen` for each target supported by Unison, see the `run-specsgen-*` recipes in `src/Makefile`.

7.6 Importing from LLVM

Most of the information required to describe a target in Unison is readily available at LLVM for the most popular targets. LLVM uses a language called TableGen to describe targets. The Unison driver for LLVM (see Chapter 3) includes an extension of LLVM's `llvm-tblgen` tool that produces a YAML file (processable by `specsgen`) with the attributes of all instructions in a LLVM target. To produce a `$TARGET.yaml` file for a LLVM target `$TARGET` (where LLVM's source code is placed in `$LLVM_DIR`), run:

```
llvm-tblgen -unison $LLVM_DIR/lib/Target/$TARGET/$TARGET.td \
            -I $LLVM_DIR/include -I $LLVM_DIR/lib/Target/$TARGET \
            -o $TARGET.yaml
```

Appendix A

Further Reading

Bibliography

[1] Roberto Castañeda Lozano. *Integrated Register Allocation and Instruction Scheduling with Constraint Programming*. Licentiate thesis. KTH Royal Institute of Technology, Sweden, 2014.